

Here is Module 8:
Modifying and Updating Existing Programs.

Up to this point, you have learned the various steps and procedures involved in developing and producing a new computer program. However, that is not the end of the programming cycle. Programs must be maintained and updated just as good tools must be maintained, and for much the same reasons.

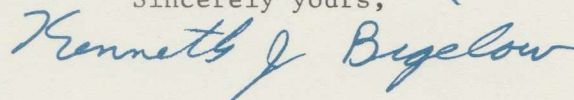
In addition, programs obtained from other sources must sometimes be modified in order to operate correctly on a computer with different hardware connected to it. Depending on various factors, the modifications required may be minor or quite extensive.

The methods required to modify or update a program depend partly on the way in which the program was coded and partly on the extent of modifications required. A compiled program may be modified slightly in some respects, but cannot be extensively modified, unless it is re-coded and recompiled. On the other hand, an assembled program can often be modified extensively in its assembled form, without a need to re-assemble the modified source code.

To help you to gain experience in studying and modifying existing programs, we have included some practical programs on your program disk, which you will enhance as you practice your new skills.

In your next Module, TOOLS AND TRICKS FOR PROGRAM DESIGN, we will show you some of the ways in which programmers simplify various phases of the programming process, from the initial design through coding and debugging.

Sincerely yours,



Kenneth J. Bigelow
Project Engineer

CONTEMPORARY PROGRAMMING AND SOFTWARE DESIGN SERIES



Welcome to **Module 8** in the Contemporary Programming and Software Design Series. In this module, we will learn a number of techniques for modifying, not only our own programs, but programs that have been purchased off the shelf as well.

Because we will be considering commercial software and techniques for modifying it, we must also deal with questions of copyrights and the law. For example, unless otherwise specified, all programs on the enclosed Program Disk (and all your other Program Disks) are copyrighted by the McGraw-Hill Continuing Education Center. This means that you may make backup copies of the Program Disk for your own use (in fact we encourage you to do so), or copy the files to your hard drive, if you have one. However, *you may not give or sell copies of the disk or any of the copyrighted programs to anyone else.* This is what the whole idea of copyright is all about.

By the same token, you may not distribute copies of anyone else's copyrighted programs, with or without charge, unless the holder of the copyright specifically permits such actions. This includes any programs that contain your modifications — you still can't distribute the program to anyone else.

As with all Program Disks in this Series, the enclosed disk is formatted for the Commodore 64 computer (and the C-128 in 64 mode). We advise that you place a write-protect tab on the program disk at once, and then use your **BACKUP** program to make a working copy for use in this Module. Remember that the working disk must be placed in

Drive 8, so that programs on the disk can find their files.

According to the Software Registration Cards we have received, Module 8 is one of the most-wanted modules in the entire Series. People want to acquire the skills and knowledge necessary to modify existing programs to suit specific requirements. We can understand this, since we often must perform such modifications ourselves in order to match our requirements, or to correct flaws that we have discovered.

You can begin this module as quickly as you like. *Begin by turning on your computer and its peripheral devices, and placing your Program Disk or its backup into Drive 8. Then, type in the commands:*

```
LOAD "WELCOME",8 <return>  
RUN <return>
```

Your introduction to tweaking, adjusting, patching, and updating programs will begin as soon as you are ready.

Contemporary Programming & Software Design

FOR THE COMMODORE 64
AND 128 COMPUTERS

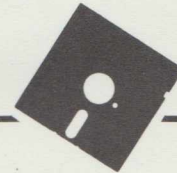


TABLE OF CONTENTS

MODIFYING AND UPDATING PROGRAMS

Introduction	1
Track 1: Things to Consider When Modifying Programs	3
Track 2: Modifying Your Own Programs	7
Track 3: Modifying Programs Where Some Information is Provided	11
Track 4: Modifying Programs without Technical Data	25
Track 5: Solutions and Problems	35

McGraw-Hill Continuing Education Center
3939 Wisconsin Avenue
Washington, D.C. 20016

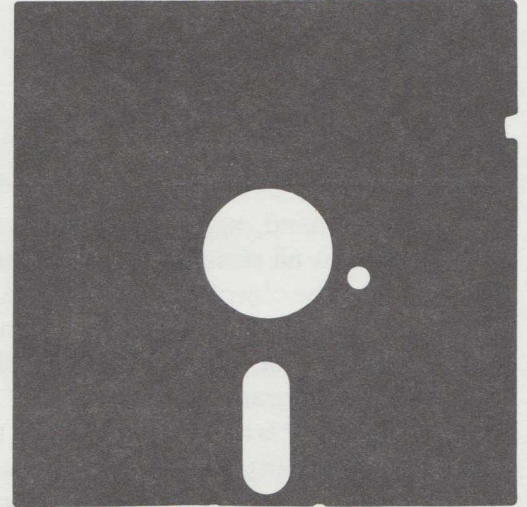
Copyright 1988 by McGraw-Hill, Inc. All rights reserved. Manufactured in the United States of America. Except as permitted under the United States Copyright Act of 1976, no part of this package may be reproduced or distributed in any form or by any means, or stored in a data base or retrieval system, without the prior written permission of the publisher.

0-07-580085-3

MODULE 8

Contemporary Programming & Software Design

MODULE 8



MODIFYING AND UPDATING PROGRAMS

There are a number of reasons for wanting to modify a program after it has been in use for a period of time. The original program author may wish to correct an error that was found after the program was released, or the author may want to add a new capability to the program. Either the original author or a user may find that the program will not work correctly on a specific computer, and will modify it slightly so that it works with the different hardware.

Minor modifications can very often be made to the program object code. Such program modification is commonly known as “tweaking” or “patching,” and the resulting “patched” program is then directly usable.

Program patching is normally performed to bypass a group of instructions or to change a single

instruction. For example, some programs intended *only* for the IBM PC and not for any of the compatibles will test for a particular sequence of bytes at a particular series of addresses in ROM. You can bypass such a test and thereby allow the program to be used on compatible computers.

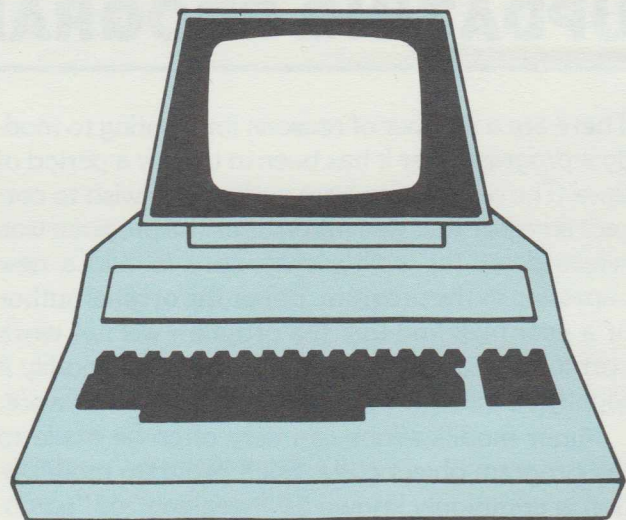
In another case, you might wish to change the nature of a conditional jump, or even change it to an unconditional jump. Or you might change the name of a command by altering a character string within the program.

All such adjustments are considered minor, because they will fit within the space of the original instructions or data in the program. You won’t be changing the size of the program, and your changes or patches are all very local in their operation.

On the other hand, major modifications should be performed, if at all possible, on the source code rather than on the object code. These are additions to the program, extensions to various commands or data, and, in general, any change that will increase the size of the program.

Unfortunately, it is not always possible to use the source code, since (if you didn't write it yourself) the author may not make the source code available. In that case, you will have to apply patching procedures on a large scale. There are several ways to accomplish this, usually by placing new code at one end of the program (either the front or the back), and then inserting JMP or JSR instructions in the main body of the code to implement the additions to the program.

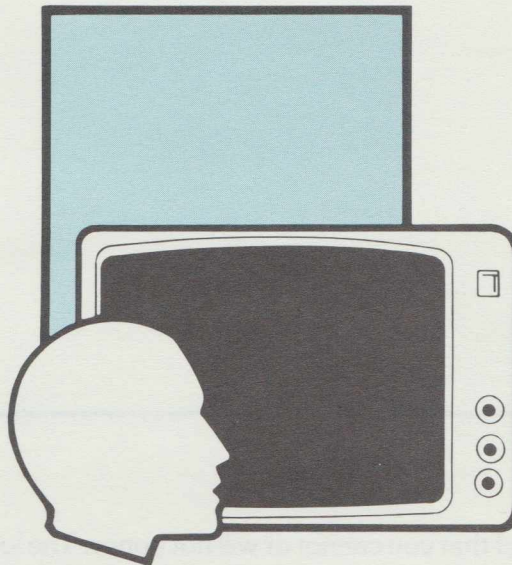
In this module we will examine a number of techniques for modifying programs. We will apply these techniques to programs where we have the source code and technical documentation, as well as to programs for which we have little or no documentation beyond the User's Manual. We will also see as much as we legally can about how to apply these techniques to commercial programs.





TRACK 1

THINGS TO CONSIDER WHEN MODIFYING PROGRAMS



Whenever you make plans to modify a computer program, there are several things you should consider before you begin, and a number of thoughts you should keep in mind as you proceed. A little planning can save a large amount of time and trouble as you modify the program.

Some of these considerations are technical; others are logical, though not technical. But the most important considerations are legal ones, so we will start with the legal considerations.

Sector 1: Legal Considerations for Program Modification

There is a substantial controversy over exactly how a copyright applies to a computer program. Clearly, if the author of a program has placed it totally into the public domain, anybody can use it in any way for any reason. Similarly, if you write a program, whether or not you copyright it, you still have the right to modify it, sell it, give it away, or otherwise decide its fate.

The question of copyright arises when you have purchased a program from either a program author or a distributor. User-Supported Software obtained either from other individuals or from computer bulletin boards also may be copyrighted, unless its documentation specifically states otherwise.

In addition to copyrights, there are license agreements that may be drawn up by the author or original owner of a program. These are formal contracts that can extend far beyond the limits of the basic copyright laws.

License Agreements. If you have signed a license agreement in the process of purchasing a program, that agreement is the contract that defines what you may or may not do with that program. As with any contract, you should read it very carefully before you sign it. Once you sign it, it becomes your agreement that you will abide by the terms of the contract. If that contract forbids you to modify the program, then it is illegal for you to do so.

If you are unwilling or unable to abide by the terms of the license agreement, *don't sign it, and don't buy the program!* Do not commit yourself to a

contract that you cannot or will not honor. The legal repercussions can be extensive and very expensive.

Boxtop Licenses. Some programs are sold with “boxtop” license agreements. In these cases, the program disk or disks are packaged in a sealed container that cannot be resealed once it has been opened. Printed on the seal is a statement that breaking the seal will be considered an acceptance of the terms of the license agreement inside. This agreement may or may not be completely visible and readable.

There is considerable controversy over the validity of these “boxtop” license agreements. In the first place, you don’t have to actually sign anything. Unlike credit cards, the use of a program leaves no “trail” to prove that you have used it, so your signature does not appear on any other documents either. Also, how can you decide whether or not to agree to license conditions if you can’t even read them before you have bound yourself to them?

Eventually, the issue of the validity of “boxtop” licenses will come up in court, and then a legal position will be established. Until then, the question has several answers, depending on your particular viewpoint. We suggest that you do not depend on “boxtop” licenses if you sell a program, since they may very well prove to be legally meaningless. Nevertheless, we suggest that you at least abide by whatever portion of such an agreement you can see before breaking the seal on any software you purchase for yourself.

Copyrights. Copyright is the basic legal protection for any original document. You can begin copyright protection for any original document you write, including a computer program, by including a copyright notice, such as the one at the beginning of this Learning Guide. Such a notice must contain the words COPYRIGHT®, the year of first publication, and the name of the copyright holder. Merely placing such a notice on the document begins your protection under the copyright laws.

As its name suggests, a copyright confers upon its holder the “right to copy” that work, and the right to distribute it for profit. Without the copyright holder’s permission, no one else may do anything that would infringe upon the holder’s right to profit from his or her work. This means that if you buy a copyrighted program, you may neither sell nor give away copies of that program. Either action will interfere with someone else’s right to sell copies of the program for profit.

However, the fact that the program author holds a copyright does not prevent you from making backup copies of the program or disk *for your own use*. Thus, you may make archive copies and backups to protect your investment against accidental erasure or damage, but only you may legally use these backup copies. You may decide to sell your original copy of the program and the manual to someone else. This is permissible. But you must either include all of your backup copies or destroy them. Otherwise, you will be in violation of the copyright laws.

Any time you buy a copy of a program, that particular copy of the program is yours, to do with as you wish (so long as you do not violate that copyright). If you wish to modify it or examine it or

just try to understand its internal workings, you are within your rights to do so, but remember that you cannot share the modified program with others any more than you could share the original program. The difference is that you may not even sell the modified program to another while eliminating all remaining backups. You can only pass along to another person the one, single, original program that you purchased.

User-Supported Software. The newest category for software is User-Supported Software. It is also known as "freeware" and "shareware," along with other names. This approach is an effort to avoid the high costs of advertising, copy protection schemes, and so forth. You can get any shareware program for only the cost of a disk and the copy service (\$10.00 maximum, usually). Or, if you have a computer with a modem, you can call up one or more bulletin boards and download such programs for just the cost of a telephone call.

User-Supported software may be freely distributed to anyone who wants a copy, without charge. Then, if you like a particular program and intend to make use of it (after trying it out thoroughly, of course), you are asked to send a small payment to the author.

The idea of User-Supported software is that programs become widely distributed throughout the computing community without any advertising. Users can test a program thoroughly before paying for it, and the price remains quite low compared with commercial software. Good, popular software

should survive, as users will be willing to pay a small fee to support the concept as well as the program. Poor programs will die out from lack of support.

With User-Supported software, as with ordinary copyrighted software, you cannot sell copies of the program to anyone else. As part of the User-Supported concept, however, you may give a copy of the program, in its original and complete form, to anyone who wants it. You may *not* distribute modified versions of the program without permission.

There is normally no objection to your modifying the program for your own use. In fact, most program authors using this distribution concept welcome comments, suggestions, and corrections to their programs. To this end, they often provide additional information, source code, etc. that you may use to modify the program to suit yourself. The only requirement is that you then keep the modified program to yourself and let the original author distribute the modifications in the next version of the program.



Thus, if you are contemplating modifying a commercially obtained program for your own private use, you are not likely to have any problems. You can even devise a separate program to modify a commercial program, and sell the patching program by itself. (There are many such programs on the market now, and patching procedures are also published in many magazines.) However, you may not distribute either the original program or the modified program in any way, unless you have been explicitly granted permission to do so. Of course, if you have signed a license agreement, you should abide strictly by its terms, regardless of any other considerations.

Sector 2: The Technical Requirements for Modifying Software

Once you have determined whether or not you are allowed to modify a particular program (and assuming that the answer is "yes"), you must consider what you have available for making changes and what you may need.

Of course, you will have the program object code to be modified. You will normally also have the User's Manual. In addition to these, you will need a debugging program such as NRIBUG (or DEBUG for PC-DOS or MS-DOS, or DDT for CP/M). Other disk operating systems have their own equivalent debugging programs. At the very least, you will need to be able to disassemble machine code to assembly

language format, and to dump out segments of a program to locate ASCII character strings, as well as to modify the contents of memory. You will also need to be able to read and write program files to disk.

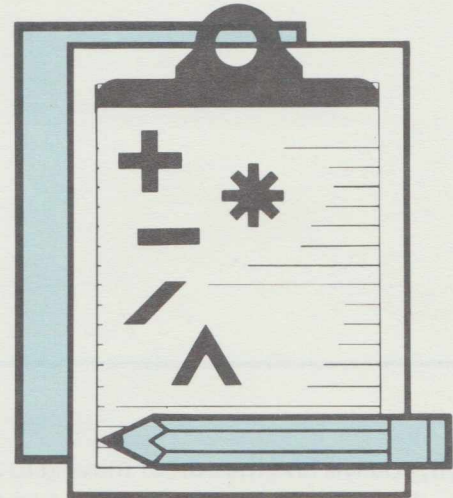
In addition to these minimum requirements, any technical documentation you may have will be highly useful. A commented source code listing will help you to locate specific functions within the program, while the technical reference manual will provide more detailed information as to how each function was implemented.

This is where the importance of complete documentation becomes apparent. Even if the program you are working on is one that you have written, if you wrote it more than six months ago you will need your own documentation to help you recall your program design as you proceed to modify it. If you did not document your work, you will have to rely on your memory for this information. Since most people cannot recall all of the details of their program design, this means trial and error, and a substantial amount of work just to figure out what you did to start with.

Along with all the documentation you can get, you should have plenty of scratch paper and an ASCII code chart. You should also be familiar with the operation of the program in its present state. This will help you to correlate program code with program action.

Once you have gathered together all available information about the program, you are ready to begin the examination and modification procedures. Keep in mind that the more information you have available about the program, the easier it will be for you to determine what modifications you want to perform and where to perform them.

TRACK 2



MODIFYING YOUR OWN PROGRAMS

If you are the author of the program to be modified, or if it was written for your company and you have been assigned to update it, you should have the complete documentation for the program available to you. In addition, you are likely to be quite familiar with the present operation of the program. You may even have some ideas of what you want to change and how you intend to proceed. In any case, modifying a program on this basis is essentially an extension of the debugging process.

There are several different classifications of program modifications, depending on what you have in mind. Each class of changes has its own preliminary requirements and procedures. Let's look separately at the primary classes of changes and see how they may be implemented.

```
LOAD "TRACK2",8 <return>
RUN <return>
```

Return to this point when your computer instructs you to do so.

Sector 1: Making Small Changes to the Program

Making a small change to a program is essentially an extension of the debugging process that you went through at the conclusion of your program design process. The only difference is that you are doing it some time after the program has been released for general use.

When you make only a slight change to a program, you are said to be "patching" that program. That is, you are applying a slight patch to the code to cover up a "hole" or error that you have detected. Changes of this sort might typically be a change in a programmed constant used within a calculation, or a change in the parameters used in calling one of the kernel OS subroutines.

These changes do not affect the size or organization of the program; rather, they adjust the behavior of the program at selected key points.

For example, you can use a call to \$FFF0 to either read or preset the current cursor location on the screen. If you mean to read the cursor location but had your program clear the carry flag instead of

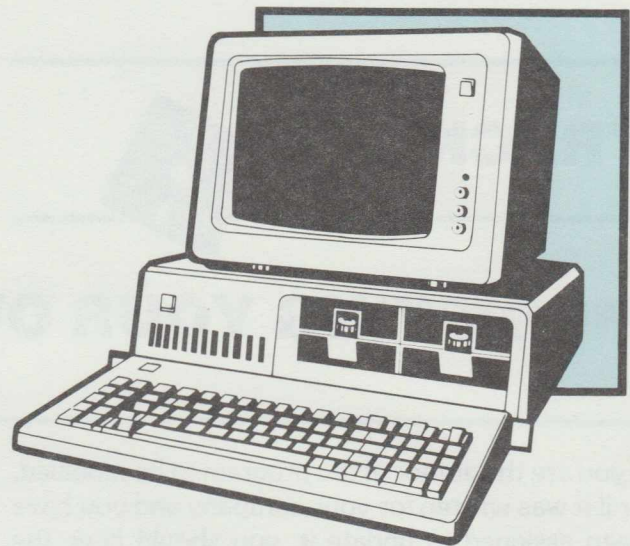
setting it, you will find the cursor moving to another, perhaps illegal, location. Or you may find that you need to adjust the values of X and Y used to preset the cursor location.

Having decided what you need to do, you can use NRIBUG to go directly to the appropriate location in your program and change the required values. You can also use NRIBUG to set the registers directly and then test the resulting operation of that part of the program. Then, when you have confirmed the change and both modified and saved the program, you can mark the change both on your source code and in your Technical Reference Manual. This is important; if you don't do this, you may come back to the program later and wonder why it does not match its documentation.

Patches of this type often involve program input/output. Another common requirement is to modify the timing of a programmed delay loop. You can also use this method to change the version number of a program, if you buried this information as a character string at either the front or back of the program.

Sector 2: Making Intermediate Changes to the Program

While small changes to a program do not affect the length or structure of that program, intermediate changes do, although to a relatively small extent. Ordinarily, you would make intermediate changes



to a program only to test the operation of the modified program. Then you would modify the source code and recompile or reassemble the program.

The advantage of “patching” a program to this extent is that you can avoid the need for several edit/assemble or edit/compile sessions. Instead, you can insert the modification and play with it directly (under NRIBUG or a similar monitor program), until it performs as desired. Then you can edit your source code once and either assemble or compile it, to confirm the change. As before, be sure to update your documentation to reflect the latest version of your program.

One way to make this type of program modification is to add a few bytes of code to a subroutine that exists within the program. For example, you might have to add a test for an unlikely condition which, if it does occur, causes the program to crash.

Or maybe you forgot to preset a register or flag before making a call to a kernal subroutine. It would be nice to be able to insert the required extra code and check it out before you go through the formal edit/reassemble/recompile process. However, this is only feasible if there is a quick way of modifying the program code without having to change address pointers and relative addresses in the process. The question is, how can we do this?

As it turns out, we can do it quite easily in many cases, as shown in Figure 2-1. We simply enter the modified code sequence *after the end of the program*, where it won't interfere with any part of the existing program. Then, we replace the original code in the program with a JMP instruction going to the added code. The added code ends with a JMP instruction back to the appropriate point in the

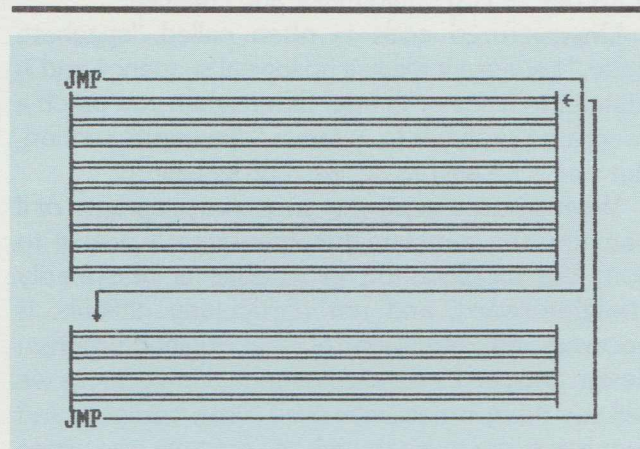
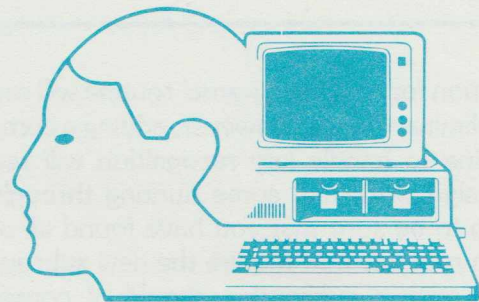


Figure 2-1. Adding code to the end of a program.



original program. In some cases, it may be preferable to use JSR and RTS instructions in place of the two JMPs. Of course, it is poor technique to leave a program like this once you have tested it and confirmed the change, but it is quick and it does work for testing purposes.

If your program does use the space immediately following itself, as a word processor or text editor will use the rest of RAM as a text buffer, there are other possibilities. Since you are using at least one disk drive, it is likely that you are *not* using the cassette buffer beginning at address \$033C. Several bytes before and after this buffer are unused by the system, so you can use the space from \$0334 through \$03FF if needed. Or, you can use the 4K (4096 byte) block from \$C000 through \$CFFF if you prefer. This block has the advantage that it will not be erased during a CPU reset, whereas the cassette buffer will be erased completely.

When designing the code to be added to the program, the amount of code to be added will determine the extent of the design work required. For example, adding a LDX or LDY instruction in

preparation for a call to a kernal routine will require little preliminary work. However, adding a complete subroutine to handle key recognition will require both design work and some hunting through the program to be sure that you have found all of the program routines that require the new subroutine.

While such a subroutine should of course be designed in accordance with all of the procedures you have already learned, the actual design work should be slight, since the modifications will be relatively small.

Sector 3: Major Changes to the Program

If you are planning to make major extensions or modifications to your own program, you should return to the initial design procedure for the original program. Design the modifications to the program from scratch; then design the connections between the original program and the new extensions. Essentially, this amounts to a redesign of the overall program, except that part of the design work has already been done.

The extent to which you should go back into the program design process is determined by the extent to which you wish to modify your program. Obviously, a slight modification will require little work, while a complete change in the way the program operates will require extensive work. The choice of how much to change the program and hence how much work to do should be based on exactly what you want to accomplish.

In order to make major changes to a program, you should be familiar with the present operation of the program. If you wrote the original program yourself, you should already be familiar with it. If a coprogrammer wrote it, you may have to spend some time getting used to it, so that you can clearly understand its present operation.

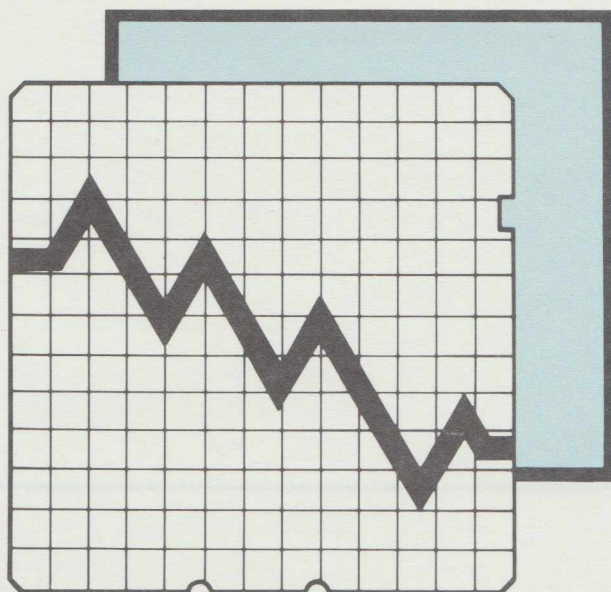
In any case, once you are familiar with the present operation of the program, you should write down the additional requirements you have for the program. State these as a new task or problem statement. Then go through the program design procedure from the problem statement you just made, through the solution steps, flowcharting, etc.

The resulting program, when it is completed, will be a redesigned program rather than a patched program. This will give it a unified structure, similar to the structure of the original program. This unified structure will make it much easier to update the program again later on than it would be if you just patched it now. In that case, you would have to hunt for various program segments and verify their functions before you could make any changes.

Unstructured code is often called "spaghetti code," because it follows no logical sequence and is difficult to trace or debug. Often when you patch a program you have to insert a "spaghetti" section, but this is no way to design a program!

When you are modifying your own programs, or if you have the complete documentation available to you, the modification procedure is reasonably straightforward, and not particularly difficult. It becomes an extension of the original program design and can be handled as such. However, as we will see next, it becomes a bit more complex and requires some extra thought if someone else wrote the program and you don't have all of the technical documentation for the program.

TRACK 3



MODIFYING PROGRAMS WHERE SOME INFORMATION IS PROVIDED

Some software is provided with its source code, or else only as source code, either with or without user documentation. For example, a number of CP/M utility programs in the public domain are supplied only as assembly-language source files, with comments and any special instructions included in the source files. This is permissible, since CP/M includes its own assembler, which, although considered slow, will do the job.

Unfortunately, this is not true of most computers and operating systems, including the C-64. You would have to buy an assembler or compiler, or use a program such as FAS, to work in any language other than BASIC. Nevertheless, many public-domain and User-Supported programs are supplied with their source code as well as any documentation. The source code makes it easier to understand the internal operation of the program so you'll know what modifications will be needed to adjust the program to meet your requirements.

The extent to which you can readily modify a program like this depends on how much documentation is supplied with the program. You can overcome the absence of good documentation in some cases, but it becomes quite difficult to generate large amounts of technical documentation without investing a substantial amount of time in studying the existing program.

With these factors in mind, let's see just how far we can go toward modifying different kinds of programs for which some technical information is provided.

Sector 1: Assembly-Language Programs with Commented Source Code

In Appendix A at the end of this Learning Guide you will find the printed assembly-language listing of a program called PDSW, for Public Domain Side-Ways. This program was written for the IBM PC and compatibles, to work with IBM compatible printers, such as the Epson. The program will take a very



wide document and print it sideways on a graphics-capable printer. In fact, it was a modified version of this program that we used to print out the wide instruction set for the 6502 microprocessor in Module 5.

Since this program was written in 8088 assembly language, it must be modified to work on the Commodore 64 and 128 computers. We have begun the process on your Program Disk, but it must still be extended and modified in several ways to operate with most Commodore printers or to be assembled by FAS. This is what we will do with this program.

The first step is to look at the documentation file, which is on your Program Disk with a filename of PDSW.DOC. To do this, first LOAD and RUN the file TYPE on your Program Disk. When it asks for a filename, type in the name PDSW.DOC.

At this point, you may choose to display the file to the screen or output it as hard copy on a printer. If you have a printer connected to your computer, you may select either option. If not, specify output to the screen. The file is small, and takes up little space.

Looking at the Source Code

Now that we have some idea of how the program works from the user's point of view, let's look at the assembly-language source code. This code is in Appendix A at the end of your Learning Guide, as



8088 source code. We also have the beginning of the source code on your Program Disk, with the filename PDSW.FAS. It is in the format required by FAS. Therefore, if you do not now have FAS operating in your computer, LOAD and RUN this program. Then, LOAD PDSW.ASM. If your printer is available, you may wish to LIST the program to the printer, so that you will have a printed listing to follow. If not, you can LIST the program to the screen, and pause the listing (we suggest using one of the function keys for this, although any key will work) as necessary to follow the discussion. Either way, refer to PDSW.FAS until it ends, and refer to Appendix A throughout this discussion.

At the very top of the listing, we find a series of defined constants. These constants will be used often within the program. Of course, it is not necessary to define such constants, but it is usually easier to recognize and understand such a symbol than it is to use an undescribed number.

We also find a group of *macro* definitions. A macro (short for macro-instruction) is a short sequence of instructions that may be used more

than once in the program. The sequence is too short to be worth turning into a subroutine. Some assemblers, therefore, permit the definition of a sequence of instructions, which will be named at the time of definition. Whenever the name is used in the program, it will be replaced with the predefined instruction sequence. Unfortunately, FAS does NOT recognize macros, so it will be necessary to expand these macro definitions by hand in the program, before FAS can perform the assembly.

The first macro defined is LPRINT. This macro is the “workhorse” of the program; its purpose is to actually output the appropriate character to the line printer. It can output a series of characters by calling itself repeatedly (this process is called *recursion*).

The next three macros use LPRINT to output characters to the printer. FORM_FEED is easy enough; it tells the printer to go to the top of the next page. SW_LF and SW_SET_BIT, however, should be examined more closely to determine exactly what these code sequences do to the printer.

We find that the sequence ESC J n causes the printer to execute a one-time linefeed of $n/144''$ or $n/216''$, depending on the printer and its operating mode. This is interesting; it is probably the reason why different printers produce different spacing between characters as they print. We will keep this in mind as we go on, and maybe we can do something about the spacing problem.

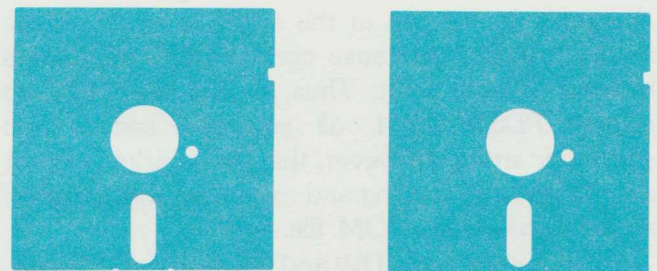
The ESC K n1 n2 sequence tells all printers to go into bit-addressable graphics mode at 60 dots per inch. The numbers n1 and n2 define the number of

graphics characters to be printed. This code sequence is the same for all Epson-compatible printers we examined.

The Start of the Program

At the beginning of the program itself, we find the assembler directives SEGMENT and ASSUME. These are required by assemblers that can produce programs that will occupy and use more than the 64k bytes of memory contained in a single segment. If the program occupies more than one segment, the assembler must know how to find each part of the program, and whether jumps and subroutine calls will be NEAR (within the same segment) or FAR (to another segment). Also, different segment registers may point to different segments at the same time, so we must tell the assembler where to expect them to point at any time within the program.

Next, we find an ORG (ORiGin) directive. This tells the assembler to start at offset 0081H within the



segment (called CSEG in the SEGMENT directive) where the operating system will load the program. The purpose of this is only to identify location 81 as the start of the filename which is to be printed out. This filename will be copied to location 81 and following locations by the DOS, if you typed it out as a parameter following the PDSW program name. This is where the program will look to find the name of the text file to print out.

Now, we begin the program itself at location 0100H. This is necessary since this will be converted to a .COM file at the end of the assembly process. The program author decided to make the program formal by defining it as a PROCedure (in this case, a NEAR procedure, since it will reside entirely within one segment). The next instruction is a JMP past some data to the actual start of the program itself.

The first thing we see after the JMP instruction is an identifying character string. The 1AH character at the end of the string is an end-of-file marker, which will cause certain DOS operations to stop at that point. *Thus, if you use TYPE to display PDSW.COM, all you will see is this character string.* However, the 1AH marker will not stop DOS from loading and executing this program as an executable .COM file.

Next, we see some DW and DB directives. These define Words or Bytes to be used by the program for various purposes. They will be set or modified by the program during execution.

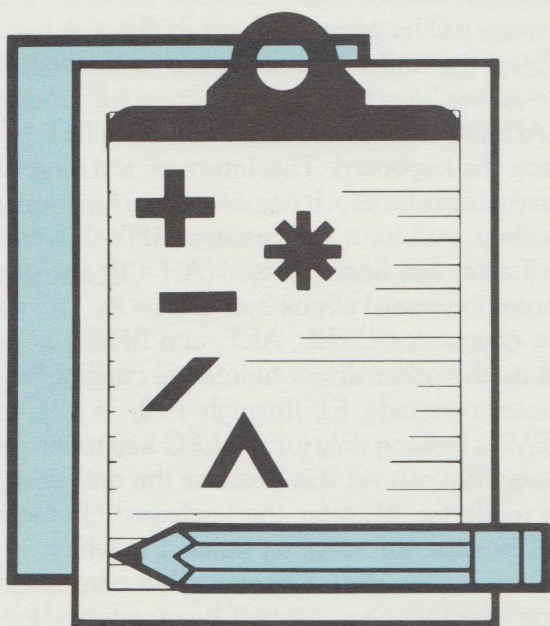
The program itself is neatly separated into functional blocks, with blank lines as the separators. The comments are somewhat limited, but they do serve

their purpose. The first block, at the label START1, is a call to perform function 9 of INT 21H. Since this is the function to type a character string to the screen, we can assume that INST is a label that defines such a string. We haven't seen that label yet, and it will probably appear at or near the end of the program. (It does; if you made a printed listing, you will find it just after the end of the program, near the start of the DB directives.) It turns out that this is the identification string for the program.

Next, we pick up the length of the filespec to be printed, and place a null (00H) at the end of it. This will allow us to use file handles rather than an FCB to open the file. Unfortunately, it also means that DOS version 2.00 or higher will be required to use this program. (Don't worry about file handles now; they are handled entirely by DOS and are easy to use. We will see how shortly.)

We want to eliminate any leading spaces, since spaces are not a valid part of any filename. This is the purpose of the loop labeled SPLP. Then we must check to see if the next character is the null we put down. If it is, no filename was typed in as a parameter, so we cannot print out a file. In this case, or if a '?' character was typed in as a request for help, the program will output a string to describe the proper usage of the program, and then stop. If we have a filename, the program will move on to NO_USAGE and will continue from this point.

The first four instructions following NO_USAGE: show the easy way to open a disk file if you have DOS 2.00 or higher. Simply point DS:DX to a character string that defines the path to that filename in any directory or subdirectory on the specified drive. The string must terminate with a null or ASCII zero, so it is commonly known as an ASCIIZ string. Set AL to 0, 1, or 2 to indicate that you wish to read from (0), write to (1), or do both (2), to the file. Then, call function 3DH of INT 21H. The function call will



attempt to open the specified file for the given purpose. If the file cannot be opened, the carry will be set upon return from the function call. If the carry is not set, the open was successful, and AX contains a *file handle* which DOS will now associate with that file for all read and write operations. The program saves this file handle for future reference and then checks for a possible error condition.

Assuming that there is no error, the program (at NO_OPEN_ERR1) outputs another message to the screen using function 9 of INT 21H, and then it proceeds to the main program loop.

The Main Program Loop

We have reached the beginning of the main program loop (at the label NO—OPEN—ERR:). All preceding instructions will be executed once only, each time the program is run. The main program loop, however, will be executed as often as may be necessary to print out the entire text file in a sideways format.

The first step in the main loop is to initialize the text buffer. As the comment explains, this is done by filling the buffer with ASCII nulls, so that no strange characters can be printed out. We then find that the ROW counter and registers DI and BX are initialized. This brings us to the label LFLP:.

We next step the row counter to indicate the row number that we are about to start filling up. We then check to see if we have already filled up 48 rows of text. If so, we must go print out this much text before we can read in any more.

If we do not yet have 48 lines of text, we go to the label NOT_YET:. We save the current text buffer pointer (register DI) and reset the COLUMN counter to zero. We must preserve BX, because this register is used to help save the pointer to the start of each line of text.

Now we come to CHARLP:. This is where we read a character from the text file. This is very easy to do when you are using file handles rather than an FCB. We place the handle number (which DOS returned when we opened the file) into BX, and place the number of characters we want to read into CX. Then we point DS:DX to the location in memory where we want the text to appear. A call to function 3FH of INT 21H will then perform the read. There is no need to predefine a DTA or point to the FCB; all of that is now handled by the DOS.

In this case, we will read only one byte, which will be a single character from the file. Upon return, AX will contain the number of bytes read, so we check it to be sure it got one byte. If it did not, the end of the file has been reached. Also, if the character read from the file is a ^Z character (1AH), it is considered to be the end-of-file marker. Either way, the program is done with the file and will terminate when it has printed out the current contents of the buffer.

If we are not yet at the end of the file, PDSW checks for key control characters (linefeed, return, tab, and form feed). Other control characters are ignored, but these are acted upon in an appropriate fashion. Then control returns to CHARLP: until the

line is filled, or to LFLP: until all 48 lines are filled or the end-of-file marker has been found.

Each character placed into the buffer (at PLACE:) is accompanied by a call in the subroutine COLWIDTH, to make sure that the 1024-character line length has not been exceeded. If it has, an error message will be generated and PDSW will halt.

Once the buffer has been filled or the end-of-file marker has been found, the program will proceed to READ_DONE. Here, PDSW first uses INT 16H to check the keyboard. This interrupt is a direct keyboard access facility. It has only three functions: get next key; wait for it if necessary (AH = 0); check to see if a key has been pressed (AH = 1); and get the current keyboard status byte (AH = 2); The status byte checks for CTRL, ALT, and SHIFT keys, as well as the other direct functional control keys. It does not include F1 through F10. In this case, PDSW is looking only for the ESC key to terminate output. We can tell this because the only comparison made for AL after the keyboard character is read is with the decimal value 27, which is the ESCape code (1BH). Since no other comparison is made, no other character will be recognized for any purpose at this point in the program.

The Output Loop

Assuming that we have not pressed the ESC key, PDSW will continue at label READ_DONE1:. Here we find that the program presets register CX with the number of graphics dots to be printed on a single character row on the printer (48 sideways LINES times 10 dot rows per character). It then calls the macro SW_SET_BIT, which was defined at the beginning of the program. This is the macro that places the printer in graphics mode at 60 dots per inch (480 dots per line, which is what we will be printing).



Next, the program initializes CX to the number of characters to be output to this print line (LINES), SI to the appropriate point in the text buffer, and a character flag to zero. This brings us to PRLP:.

This is the point at which PDSW actually checks the character, converts it to a series of graphics rows, and outputs them to the printer. The first requirement at PRLP: is to replace any null characters with ASCII spaces. If the character is not a space or null, PDSW will go to NOTZERO: and set the two indicated flags. Either way, it will then fall through to PRINTIT:.

At this point, PDSW converts the ASCII character to a sequence of graphics characters. We first subtract 32 (hex 20) from AL, so we can eliminate the control characters. We multiply the result by 8, since there will be 8 graphics rows per character. (The other two are blanks, for separation between characters. We'll see that shortly.) The resulting product in AX is used as an offset into the table (SWBIT) of graphics characters. DI is pointed to the appropriate character sequence, and CX is initialized to count off 8 characters. This brings us to CXLP:.

At CXLP: we find a series of four instructions that form a loop. These instructions are used to print out the eight successive graphics bytes that will form the selected letter in a sideways format. After the eight bytes have been sent to the printer (using the macro LPRINT within the loop), the program sends two nulls to the printer to provide the sideways line spacing.

Finally, SI is stepped backwards to the matching character in the next line to be output, and the program loops back to PRLP until this character position has been output for all 48 lines on the page.

At this point, we need to advance the paper for the next character column in the sideways output. This is done by the macro SWLF. Now, this is where we had a problem with some printers. We note that this macro is used here, and we will continue to check for it through the remainder of the program. We can consider how to change it to make it more universal after we see all the ways in which it is used.

Since the program has at this point completed one column of characters printed sideways on the paper, it checks its character flag to determine whether or not it needs to pass a formfeed character to the printer, or just move on to the next character column. A formfeed will be performed after all columns of a given set of 48 character rows have been printed, and at the end of the file.

Following the formfeed (at NEXT—PAGE:), PDSW checks to see if the end of the file has been reached. If not, it goes back to NO—OPEN—ERR to read the next 48 lines of the file and print them out. If the EOF marker has been reached, however, PDSW will move on to ALL—DONE: and will terminate at that point. Here, the use of function 4CH of INT 21H is preferred to INT 20H or function 0, since function 4CH permits AL to pass a code back to the calling program in AL. This code is used in this case to indicate that no error occurred during the execution of the program.

The End of the Program

Following the main body of the program, we find a subroutine defined as a NEAR PROCEDURE. This is

LPRINT1, which is the routine that actually outputs everything to the printer. It uses INT 17H, which is the direct printer function call. This interrupt has three functions, depending on the contents of AH. They are: Print the character in AL (AH = 0), Initialize the printer (AH = 1), and Read the printer status (AH = 2). In each case, DX contains a value from 0 to 2, depending on whether LPT1, LPT2, or LPT3 is to be used. In the case of this subroutine, the printer must be LPT1 since DX is cleared by the XOR instruction.

Next we find some of the character strings that will be output by PDSW under various circumstances. These are followed by the ERROR routine. This routine will be called either at ERROR: or ERROR1:, depending on the particular error detected. It uses function 9 to output the selected error message, and then function 4CH to terminate the program. Note in this case, however, that AL will contain 01 rather than 00. This is the return code that will be passed to DOS or any calling program. It can be used, if desired, to flag the fact that an error occurred, so

that the calling program can take appropriate action.

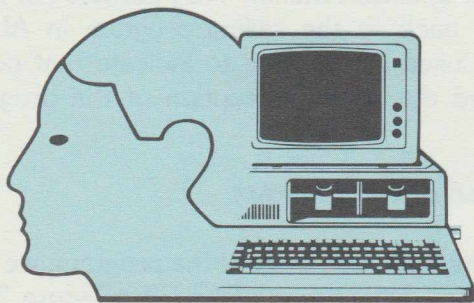
Following the error exit routine, we find the actual data for the individual characters that will be printed by the program. The first line corresponds to the ASCII space character, since the PRINTIT routine subtracted 32 from the actual ASCII character code.

After the graphic character data, we find the label LINE_PTR. This label identifies the assembler directive LABEL, which tells the assembler to identify LINE_PTR as a program label without allocating any program space. The word WORD following LABEL forces the assembler to assign LINE_PTR as a 16-bit word label rather than a byte label.

Next, we find USAGESTR, which will be printed out only if PDSW cannot find a filespec when it is called. In fact, PDSW will overwrite part of this character string as it records pointers to each of the 48 lines to be output. This is permissible; PDSW will overwrite the string only if it doesn't need it.

Then, the symbol TEXT_BUFFER is defined, so the program will be able to find this buffer, which will start 48 words beyond LINE_PTR. In fact, the text buffer will begin somewhere in the middle of USAGESTR, but, again, this will cause no problems since the two will not both be used during any one run of the program.

The last three lines of the program provide clean ends to the initial directives that appeared right after the macro definitions. ENDP defines the end of a named PROCedure (START in this case). ENDS defines the end of the SEGMENT definition for CSEG. Finally, END tells the assembler that this is the END of the program. By placing the name START as the parameter for the END directive, the program author has told the assembler that the program is to begin execution at the location designated START, which in this case is at offset 0100H.



Before you make any plans to modify an assembly-language program, you should be sure that you understand its operation in this fashion and to this extent. If you do not understand how the program works, you cannot modify it meaningfully.

Sector 2: Modifying the Assembly-Language Source Code

The first step in modifying this program for any purpose is to convert it to 6502 assembly language, and use the I/O calls provided in the Commodore computer rather than the function calls provided by IBM's DOS. The `SEGMENT` and `ASSUME` directives are meaningless in the C64 or C128, as is the `PROCEDURE` directive. Also, Commodore does not use file handles, so this storage location has no purpose and can be eliminated. Instead, our modified program must specifically request the file to be printed in sideways format, open that file, and read its contents.

In addition, the IBM version of the program is written for IBM-compatible printers, which means that they will be compatible with IBM's character set and the Epson printer control codes. This can still work with a Commodore-compatible printer interface driving an Epson-compatible printer, but will not work properly with a Commodore printer or compatible. To make it work, we will have to adjust the printer control codes sent out by the program.

One further point on printer compatibility: Commodore printers will have more graphics characters and fewer letters, numbers, etc. For



example, the normal Commodore display contains only upper-case letters, with a lot of special graphics characters. To match this, the graphics table starting at `SWBIT` must be modified to produce compatible characters.

The file `PDSW.FAS` on your Program Disk contains the initial constants and functional definitions required to implement `PDSW` on an *Epson-compatible printer*. This includes redefining the macros from Appendix A in terms that `FAS` can handle, and setting up for all three passes that `FAS` will make through the actual source code.

The original program allowed for 48 sideways lines of text, with up to 1024 characters per line, per sideways page. However, this requires up to 48K bytes of memory to be used at any one time. We don't have that much memory available in the C64, and some of that memory will be required for the program. However, we do have about 39K of memory available, and the program won't need too much space. Therefore, we can probably use 24K of space at a time for the text file. This will limit us to 512 characters per sideways line, but that should not be too much of a problem in practice.

*If you have not already done so, **LOAD** and **RUN** your copy of `FAS/MV5`, which was provided on your Program Disk for Module 5. Then **LOAD** the file `PDSW.FAS` from the enclosed Program Disk. We will now continue to convert `PDSW` from IBM format to Commodore format, so that it will operate on your computer.*

LIST PDSW as it stands now. You will see that the preliminary definitions have been made and that we have started the actual assembly loop, using PASS as the assembly loop control variable. Next we find a SEND command. This command produces a structure that will be accepted by BASIC as (in this case) a valid SYS command on Line 10 of the program being generated. In fact, the SYS command that is executed when you RUN FAS was generated in exactly this way. The UNSEND command completes the BASIC program line, so that it will appear as a completely valid SYS command to BASIC.

The text string that appears next is a modification of the one in the original IBM program from Appendix A. While it is true that we are modifying the program substantially, it is also true that Donovan Kuhn wrote the original, and should be given proper credit for it. Failure to observe this kind of courtesy will rapidly result in a bad reputation with other programmers and users.

Following the character string, we find space for the variables we will have to maintain in the Commodore version of the program, and then the actual beginning of the working program at START. Now, to complete the program, we will have to convert the remainder of the IBM version to its Commodore equivalent.

At START1 in Appendix A, we first find a set of instructions ending in INT 21H. Now, INT 21H is the general-purpose function handler in PC DOS/MS DOS. The number in AH defines the function to be performed, and other registers contain whatever

other information may be required. Function 9 displays a character string on the screen. The address of that string is placed into DX before the function call is made. Thus, these three instructions have the effect of displaying the character string at INST on the screen. Looking near the end of the program, we find INST a few lines above the data table. It simply identifies the program. We will modify this string to indicate the changes we make in the program, but we still want to output a similar string. Therefore, we will retain the program sequence at START1.

Of course, the C64 kernal routines are all character oriented; they do not include a means of outputting a character string as such. However, there is no reason we can't design a subroutine to do this job. In fact, we'll probably have to design several. Now, we are starting the program itself at line number 1000, just to keep things neat. All lines before this point are preliminary to the program, and need not be executed during the assembler passes. To leave plenty of room, and to keep the program structure orderly, let's start the subroutine at line number 10000.

There are several ways in which we could set up a subroutine to output one character at a time to the screen or to any I/O device. However, if we think ahead, we may be able to set it up to be able to output *any* character string or data string, either to the printer or to the screen. If we can do it that way, we can save ourselves a lot of trouble designing a similar subroutine just for the printer. So what will we need?

Since many different strings will be used, the subroutine must be told where a string is in memory. Strings may be of any length, so the subroutine must either be told the length or else be given a way to sense the end of the string. However, the graphics

data for the printer may have any coded value at all, so we can't use a special code to mark the end of a string. Therefore, we will provide the length of the string to the routine.

To locate the string in memory, we will need to use indexed addressing, with the address in a Zero Page location, indexed indirectly through Y. The string length can be placed in X. Then the calling routine can either place the address of the string in A and Y, or else place it directly in the Zero Page. The subroutine is a little simpler if we use the latter method, but is not as consistent with the operation of the kernel. In fact, to be completely consistent, we should set X and Y to the address, and A to the length of the string. Therefore, we will do it this way.

The subroutine will *not* be expected to open or close any device or file. It will simply output <A> characters to the current device, starting at address <YX> (Y contains the high-order half of the address). Therefore, the subroutine must:

1. Save X and Y in the Zero Page location it will use.
2. Copy A to X for the count.
3. Clear Y.
4. Perform LDA (ZP),Y.
5. JSR to the CHROUT routine in the Kernel.
6. INY
7. DEX
8. If X is not zero, go back to Step 4.
9. RTS

At this point, we could flowchart this routine. However, the above description is direct and

straightforward; we can write the code directly, as soon as we choose the Zero Page locations to use. To do this, we first note that BASIC will not be running, so we need not worry about its use of the Zero Page. Also, we will have our program reset BASIC's pointers before it exits, so we can use whatever we like. However, if we avoid possible conflicts, we can help prevent unforeseen problems. Let's use \$FB and \$FC for the pointer address. The resulting subroutine, starting at line 10000, is:

```

10000 STROUT:    STX $FB:STY $FC
10010           TAX:LDY #0
10020 STROL P:   LDA ($FB),Y
10030           JSR CHROUT
10040           INY
10050           DEX
10060           BNE STROL P
10070           RTS

```

Strings are limited to 256 characters each, but this suits the internal architecture of the micro-processor perfectly, and should not be a problem for us. If we do need to output a longer string, we can simply break it up into shorter ones that we can handle.

With the subroutine in place, we can now insert the code at START1 to call it, to output the initial message:

```

START1: LDX #FN L(INIT):LDY #FN H(INIT):LDA
        #INITEND-INIT JSR STROUT

```

We will define INITEND to be the address immediately following the character string, when we actually write the modified string.

The next step, as seen in Appendix A, is to get the filename of the file to print out in sideways format. In the IBM PC, that filename is already present in



memory, but for the Commodore computers this is not possible. Therefore, our program must request the filename, read it from the keyboard, and then actually open the file.

The INIT string can easily be extended to include a request for the filename to be printed, so the next step is to input the filename from the keyboard. We will use the text buffer area to hold the filename long enough to open the file.

We will use the same Zero Page location as our address pointer that we used with STROUT, since there will be no interference between these parts of the program. Therefore, we can set up a program

loop to read the keyboard until the <return> key is pressed:

```

                                LDA #FN L(BUFFER):STA $FB
                                LDA #FN H(BUFFER):STA $FC
                                LDY #0
KBRDLP:                        JSR CHRIN
                                STA ($FB),Y
                                INY
                                CMP #CR
                                BNE KBRDLP
                                ;-----
                                DEY
                                TYA
                                LDY #FN H(BUFFER)
                                LDX #FN L(BUFFER)
                                JSR SETNAM
                                ;-----
                                LDA #8
                                TAX
                                LDY #$FF ;MODIFY FOR
                                    SECONDARY COMMAND
                                JSR SETLFS
                                ;-----
                                JSR FOPEN
                                JSR READST
                                BEQ OKOPEN

```

Here, the first section of the routine reads characters one at a time from the keyboard, and stores them in the text buffer. The second section uses the character count in Y (decremented to drop the CR character) and the address of the buffer to set the name of the file to be opened. The third section sets the logical device numbers, while the last section opens the file and checks for an error. If an error occurred, your program should display the appropriate message and then quit.

Sector 3: What Kind of Modifications to Make in Assembly Language

If your printer requires a specific secondary command or code to enable graphics mode, place that code into Y just before the JSR to SETLFS. A value of \$FF in Y will override the secondary command or code.

At this point, you are almost ready to complete the conversion of this program on your own. However, as you scan Appendix A, you will have to look for a few additional factors that are specific to PC DOS/MS DOS. These are:

1. Function 3DH of INT 21H will open a file.
2. Function 3FH will read a character from the file.
3. Function 4CH will terminate the program.
4. INT 16H will read one character from the keyboard. It is used here to check for the ESCape key. You can use the Kernal routine to look for the STOP key instead, to terminate the program at once.
5. INT 17H sends a character to the printer.
6. Registers SI, DI, and BX are generally used as index pointers. CX contains a character count.

When you are ready, proceed to convert PDSW to Commodore code. Follow the comments more than the actual IBM source code; they convey the idea of the program clearly, and will simplify your task.

When you have the assembly-language source code and an assembler program, you can make any and all changes you may wish to the program. There is a point, however, beyond which you will essentially be writing your own program and incorporating some of the techniques from the original version. You must draw the line for this distinction yourself, according to your own requirements. However, there are some guidelines that you can follow to determine whether you should approach a program intending to modify it or to rewrite it to a major degree.

For example, if your purpose is to see how PDSW reads the name of the file to print, opens that file, and reads text from the file into memory, you would be silly to try to modify the remainder of the program to perform the task you have in mind. Rather, you should incorporate only that portion of PDSW that is of interest to you into your own new program, with modifications as appropriate.

On the other hand, suppose you wanted a program that would print text sideways in two working text columns, as on facing pages of a small program or bulletin. In that case, you could readily use the existing PDSW program as a basis for your new program design and simply add your own extensions and modifications as necessary.

As another example, suppose you wanted to change the character size, or perhaps make the printed characters near letter quality. These requirements can be met basically by changing the printer output routine and/or the character graphics data table. The same kind of logic holds true if you want to graphically print out characters that are two or more print rows tall or that are extra wide.

As a general rule, if you have a program that performs the general function that you want, but you wish to change either the way it performs that function or the options available, then you should seek to modify the existing program. However, if you wish to employ a particular technique to perform a different kind of task, you should write your own program, or else look for an existing program that more nearly fits your requirements.

Sector 4: Modifying Compiler Language Programs

If you have a program that was written in Pascal, C, FORTRAN, or any other high-level language, you can modify the source code as you wish, just as you could with assembly language. But you will need to have a compiler for the appropriate language in order to compile your modified program, just as you would need an assembler program to be able to assemble the modified assembly-language source code.

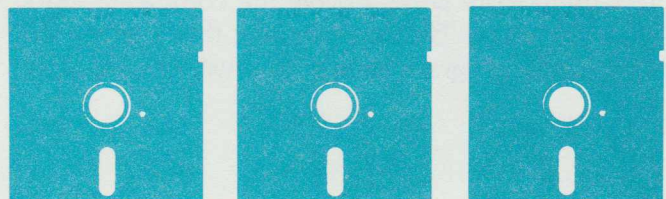
If you do not have the appropriate compiler or assembler, you can design the desired modifications on paper, using your printed copy of the source code. However, all modifications would have to be made by hand, using NRIBUG or a similar utility, just as if you had no source code listing. We will examine this kind of program patching in Track 4.

The only real difference between modifying compiler-language source code and modifying assembly-language source code is the nature and

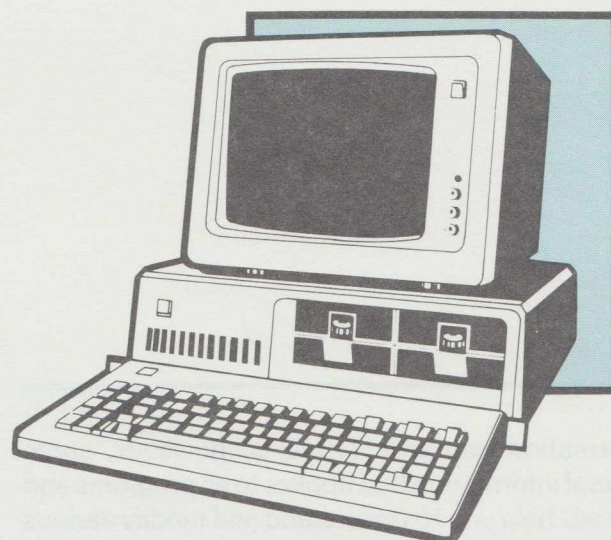
structure of the source code itself. In each case, the source code is placed in an ASCII text file on disk, and it can be edited in any desired fashion by either a text editor or a word processor. Some compilers include a text editor function so that you can edit and recompile a program without having to load and reload the editor and compiler alternately. Most versions of FORTH include at least a basic line editor, and Turbo Pascal (TM Borland, Inc.) has a WordStar-like editor. Some other compilers may require you to edit your source code separately.

In any case, the process of locating the portions of the program you wish to change is essentially the same for any language. A general understanding of the overall program is gained most easily from a printed copy of the source listing. This allows you to follow the flow of program logic and to identify the purpose of each line or block of code.

On the other hand, the process of locating calls to the kernal routines or to specific devices is best performed from NRIBUG, to locate key code sequences. The most effective way to locate and identify specific program functions and sequences without having to understand the entire program first is to use NRIBUG and the printed listing together, and watch the program in actual operation.



TRACK 4



MODIFYING PROGRAMS WITHOUT TECHNICAL DATA

Modifying a program without the benefit of a technical reference manual or any source code is a difficult — but not impossible — task. There are many approaches you can take and many procedures you can use to simplify the process.

Of course, experience helps in this sort of work, just as with any other task. But everyone has to start somewhere, and it is possible to gain the experience you need just by practicing with existing programs. The more you work with programs in any capacity, the more you will understand them and the more you will be able to manipulate them. As you gain that experience, you will find yourself working more and more easily with other programs.

Of course, as with any program modification, you should always work with a backup copy of the program. That way, if the copy totally self-destructs during testing, you still have the original program and can start over again from the beginning. If you have a copy-protected program, so that you cannot make a backup of it, you should not attempt to modify it until you have gained some experience with other programs. Even then, you should keep a

write-protect tab on the diskette until you are ready to actually write the modified program back to disk, and (if possible) write the modified program back to another disk. At least write it back under another program name so that you do not erase the original program.

At the very least, write down all of the changes you have made and the original program data and instructions before you make the changes. That way you will be able to restore the program to its original form if necessary.

Sector 1: Basic Information Required when Modifying Programs

Whenever you set out to modify a program, you should have as much information on hand about that program as you can. With a commercial program this is normally limited to the User's Manual, which contains little or no technical information about the program's internal workings. Therefore, you will have to decipher the technical

information yourself. There is, however, some general information that applies to all programs and that will help you to understand and modify various kinds of programs.

First, keep a list or manual detailing the various system calls available to the programmer. In ROM, these will be calls to the Kernal, through its jump table at the top of memory. *Even if you have the complete source code of the ROM, do not use any other entry points to ROM.* The reason for this is that Commodore may at any time modify or upgrade their ROMs. They will leave the jump table in place, but the internal routines may be moved in different versions of the ROM.

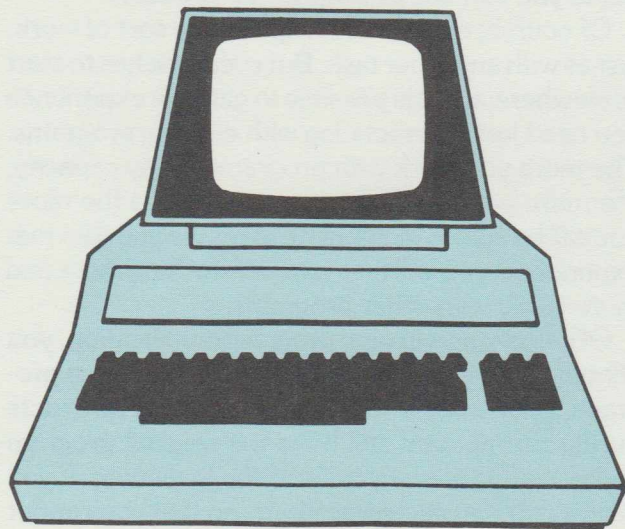
If you use the DOS wedge, know its capabilities and available commands as well, so that you can recognize any access to it.

The next thing to keep is a list of all I/O devices attached to your system, and all means of accessing them. Keep your list of device numbers and capabilities up-to-date and complete. One of the most common reasons for wanting to modify a program is to allow it to work on a computer that is similar to but not identical with the computer it was written for. Conversion from one type of computer, such as we practiced in Track 3, is also a common requirement. Either way, you will need to keep track of all differences between the two computers involved.

Third, obtain a Technical Reference Manual for your computer. Commodore sells a *Programmer's Reference Guide* which provides substantial infor-

mation on the structure and organization of the computer and its memory.

In addition to these items of technical information, you should have a printer connected to your computer and be ready to use a substantial amount of paper. You should not make any modifications to a program the first time you look at it. Rather, you will use various techniques to distinguish program areas and data areas from each other, and to identify any character strings. You will normally print out all of this information and study it before going back to make any changes to the program.



Sector 2: The Preliminary Work

Before you begin to consider what modifications to make, you must understand the program in its existing form. There is a certain amount of preliminary work to be done before you make any program changes.

The first step is to load the program along with NRIBUG or a similar debugging utility program. The utility must be capable of dumping out the contents of memory in hex and ASCII format, and of disassembling machine code into assembly-language mnemonics. It should also be capable of tracing through program execution one or a few instructions at a time, and of modifying memory. We will assume the use of the MON function of FAS, which is basically the same as NRIBUG, for the purpose of this discussion.

Once you have loaded the program, use MON to examine its general structure. Determine first if this is a compiled program rather than an assembled one. A compiled program will contain a large block of predefined subroutines, and will have an execution address well past the start of the program in memory. Often, the program will consist almost exclusively of JSR instructions to these subroutines, together with embedded character strings and in-line constants. Modifications to this type of code are necessarily limited in extent, although not impossible. Also, such programs are usually difficult to decipher.

In any case, if you plan to proceed with your modification efforts, the next step is to dump the

program in hex/ASCII form to the screen. Use this dump (the M command) to locate and identify any character strings, large blocks of a single value (often 00), and any other data you can recognize as such. This includes graphics characters that may be part of the program's screen display.

As you identify each block of such data, write down its beginning and ending addresses. You know that this is probably program data, rather than program code. You will not need to disassemble these parts of the program. There may still be data embedded in the program, but at least you will have found most of the non-program material.

Now use the M command again to dump out those program segments you have just identified. However, this time use the O command to open device #4 (your printer). This will output the dump to the printer, so that you will have it in hard copy for later study. When you have completed the printout, use C to close the printer device.

Next, use the D command to disassemble the actual program code. Skip over the data areas that you previously identified. You may still find some impossible instructions that will show up as ??? marks in the display. This means that this is not a valid machine instruction, so it must be data instead. Such data is often placed immediately after a JMP or RTS instruction.

When you have the program areas defined, disassemble the actual program code to the printer, to again have a hard copy of the program to examine as you scan through the program in memory. This listing will have no labels, no comments, and no symbolic names. Nevertheless, it will be easier to work with than a plain hex dump of the machine code.

Next, use the U command to unassemble the program. Skip over the data areas that you have

already identified. There may be individual sections of data, and they will usually show up as ?? marks in the mnemonics column of the output on the screen. This means DEBUG can't understand the instructions as valid op codes, so they are probably data. If you find such additional data, identify the starting and ending addresses and dump the data out to the printer as well. Such data is usually easy to find in an unassembled listing; it generally follows immediately after a JMP or RET instruction.

Next, unassemble the program again, this time using the ^P function to dump the listing to the printer. The resulting assembly-language listing will not have comments, labels, or any symbolic names. Nevertheless, it will be much easier to work with than a plain hex dump of the machine code.



Sector 3: Deciphering the Assembly-Language Listing

First, take a ruler and a pencil. Go through the program looking for subroutine calls and JMP instructions. Draw a line underneath each JMP instruction to indicate a break in program flow. In the same manner, draw lines to isolate subroutines from each other. RTS instructions indicate the end of a subroutine, while JSR instructions elsewhere in the program will identify the entry points to the subroutines.

After you have identified the subroutines and isolated them from each other, start looking for JSRs to the Kernal and to other utilities that may be present. These can be very helpful in relating program code to program operation. For example, if you find a JSR to address \$FFE1, the program is checking for the STOP key to be pressed. Since the Z flag will be set by this Kernal routine if the STOP key was down, this call will normally be followed by a BEQ or BNE instruction. Or, if you find a loop that includes a sequence such as LDA adrs,Y:JSR \$FFD2:INY:DEX:BNE to the LDA instruction, it is By looking at the hex dump starting at "adrs," you can often determine the purpose of the routine or subroutine at a glance. Even if you cannot tell that purpose yet, write the string down to the right of the program code so you will have it there from then on. If the program reads the keyboard at any time, look for the JSR \$FFCF call. The following code will tell you just what the program does with the character(s) it receives. Individual subroutines, especially the shorter ones, can be deciphered readily and their functions defined. Each time you recognize the purpose of any block of the program,

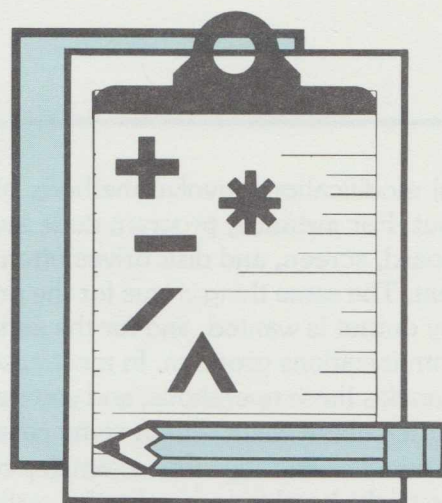
write it down on the program listing. Also, when you have determined the purpose of a subroutine, write that purpose down at each call to that subroutine. This will make it very much easier for you to follow the program flow later on.

As you progress with the deciphering and filling in of individual functions and operations, you will find that larger blocks of the program also begin to make sense. As this occurs, write down the purpose of each program segment that you can identify. Eventually, you will have your own comments marked on the unassembled listing to describe the operation of the entire program.

In addition, as you proceed through the program listing and decipher it piece by piece, you will find that you understand the operation of the program more clearly than you would just by looking at a commented listing, without some extra study. The mere fact that you have been working to decipher the operation of the program constitutes a rather intensive study of that program. Therefore, by the time you have completed deciphering the operation of the program, you will be in a position to know what kinds of changes you want to make in it.

Sector 4: Selecting the Modifications to be Made

Once you have a commented listing, you can proceed very much as if you actually had an assembly-language file on disk. That is, you can determine



where and how you wish to make changes. However, this is not really an assembly-language program, so you do have to live with a few limitations.

Since your unassembled listing does not constitute complete assembly-language source code, you cannot use an assembler to assemble it directly. Of course, it is possible to type the listing using a text editor or word processor, and thus to produce a true assembly-language source file. However, for a large program this is usually far too much effort. You will normally be limited to making patches to the existing program so that no change can occupy more space than the original program code. You can shorten a routine or subroutine, or you can keep it the same length even if you change it, but you cannot lengthen it unless you use some of the unusual techniques that we will cover later on.

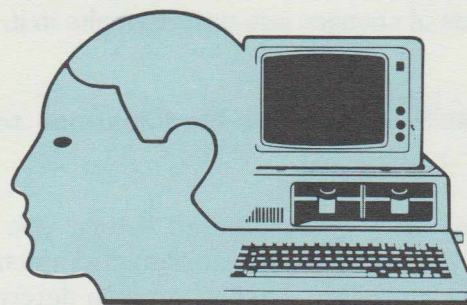
Typical modifications involve methods of input and output. For instance, program code involving the keyboard, screen, and disk drives often needs adjustment. The same thing is true for the printer if hard copy output is wanted, and for the serial port for a communications program. In most cases, the system handles these operations, and you need not be concerned about them. But in some cases they must be handled directly. For example, clocked interrupts might be able to handle the serial communications port at 300 baud (bits per second), but they cannot keep up at 1200 or 2400 baud, both of which are coming into common use. And at 9600 baud, which is being used to some extent and will become more popular as the technology improves and costs go down, clocked interrupts don't have a chance. The communications program must watch the serial port *constantly* so it won't drop any data. For this reason, modifying communications programs involves changing the port address of the serial port and possibly changing the way it is handled. Modifications are never used to make the program universal among different computers. This is unlike other types of program changes.

Another factor to watch for is the address of the video display memory. If the program accesses the display memory directly, it will not work on computers where the display memory is elsewhere. For example, in the IBM PC, the monochrome display memory starts at location B000:0000 (or absolute address B0000H). The color display memory begins at B000:8000 (or absolute address B8000H). However, in the Sanyo MBC-550 series computers,

which use MS-DOS, video display memory is in three banks, arranged by color. Thus, the red display occupies 16K bytes starting at F000:0000, and the blue display occupies the next 16K (starting at F000:4000). The green screen memory may be placed almost anywhere in the bottom 256K and is normally selected to occupy the top 16K of available RAM within this range.

Because of variations like this, different systems use different codes to handle their video displays, and a program written to write directly to the screen of one machine may be useless on another. Therefore, it is not uncommon to modify a program so that it uses system function calls instead of reading and writing the screen memory directly.

Speed of execution is the primary reason for reading and writing the screen memory directly. It takes considerably longer for the Kernel systems to determine where a character should be sent, or to place it at the correct location on the screen, than it does to execute an STA instruction. However, most programs operate rapidly enough without direct access, so many users would rather make their programs universal to all compatible computers, not just those using a specific screen orientation and memory location.



Except for communications programs and their need to directly access the serial communications port, I/O should only rarely be performed directly. After all, the computer will have to wait for the printer or for a user response from the keyboard anyway, so increasing the program's execution speed makes little sense. A similar argument holds true for disk I/O; the serial bus operates at a fixed data rate, so direct handling will serve no useful purpose. Bypassing the Kernal routines will only make your program more complicated.

In addition to methods of I/O handling, of course, you may wish to add modifications of your own to a program, according to your own requirements and preferences.

Sector 5: Typical Addresses Accessed in the Commodore Computers.

Although a program may access any address within the entire 64K byte addressing range of the 6502 microprocessor, you will find that in practice only a relatively small number of addresses will need to be checked or modified for any particular application. Similarly, in theory, any address may be used for any purpose. In practice, however, specific areas of memory are allocated to specific functions, and you can recognize them directly.

The 6502 does not recognize I/O ports as being separate from normal memory, although some microprocessors do keep I/O ports separate. Nevertheless, the Commodore computers do require a series of I/O ports for different purposes, and keep them in a particular section of memory.

As we mentioned earlier, it is considered poor technique to access I/O ports or program memory outside your own program, except for calls to the Kernal and necessary specific accesses. Most functions are better handled through the Kernal. Nevertheless, some programmers do try direct access, to make their programs run a little faster. Unfortunately, if your computer has a different hardware or I/O configuration, the program will not work properly. In fact, it may not work at all. One of the primary reasons for wanting to modify a program is to make it work on a different system than the one for which it was designed. To this end, you should recognize the general memory map in Figure 4-1 (on page 32), as well as the following groups of addresses, which have specific purposes and uses within the Commodore computer system:

1. Addresses 0000-0001. The C64 and C128 use a variation of the 6502 microprocessor, known as the 6510. The instruction set is the same, but it also contains an 8-bit I/O port. Address 0000 is the direction register for this port, while 0001 is the data register. This port handles the cassette I/O and also controls the memory map of the system by determining which ROMs will be enabled and which disabled.
2. Addresses 0002-03FF. This part of RAM is used by the system for many purposes. This includes pointers, temporary storage, the cassette I/O buffer, and a number of vector addresses that may be altered for specific purposes.
3. Addresses 0400-07FF. This is the video display RAM. Actually, the 25 line $\times$ 40 column display requires addresses 0400-07E7. Addresses 07F8 through 07FF are used as sprite pointers when needed.

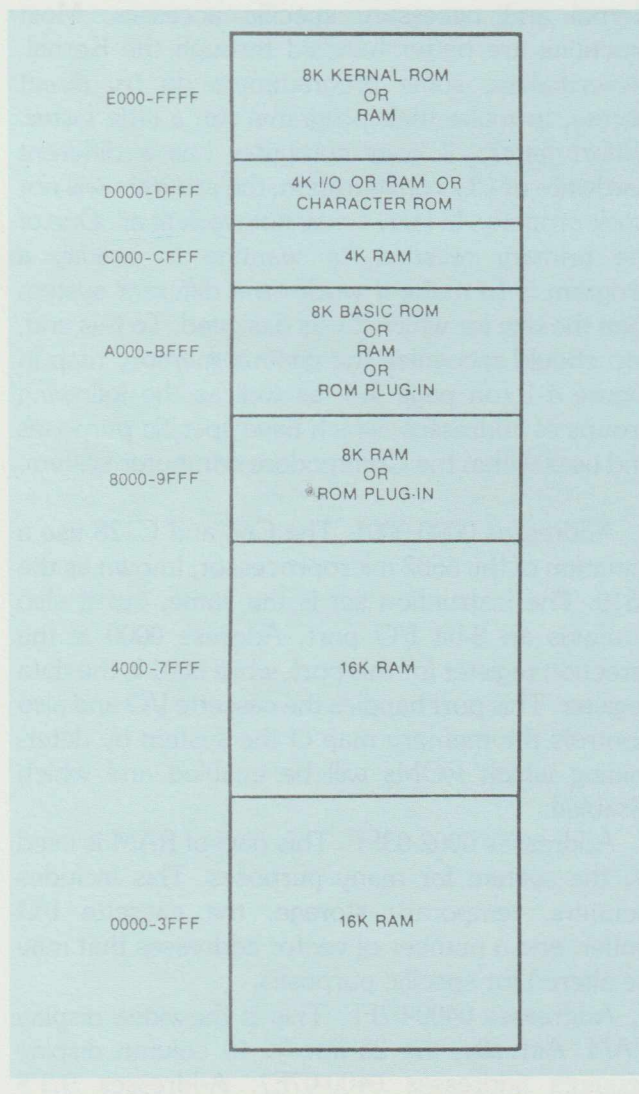


Figure 4-1. Commodore 64 memory map.

4. Addresses 0800-9FFF. The normal BASIC program space. However, if a ROM cartridge is installed, it occupies addresses 8000-9FFF, and takes precedence over BASIC. Then it can use the rest of this space as necessary.

5. Addresses A000-BFFF. 8K BASIC ROM. If bit 0 of address 0001 is 0, the BASIC ROM is switched out and this space becomes user RAM.

6. Addresses C000-CFFF. 4K user RAM. Available for any purpose.

7. Addresses D000-DFFF. I/O devices, color RAM or Character Generator, or else 4K user RAM. Bit 2 of address 0001 selects the character generator ROM (if 0). Figure 4-2 on the next page shows the normal I/O configuration of this space.

8. Addresses E000-FFFF. 8K Kernal ROM. If bit 1 of address 0001 is 0, the Kernal ROM is switched out and this space becomes user RAM. However, if you do this the Kernal service routines will no longer be available to your programs.

Sector 6: Locating and Changing Specific Code in the Program

Now that we have an idea of what sort of things might have to be changed in both commercial and public-domain programs, how do we find them in the code? How can we be sure that we have found all necessary references? It would be a substantial job searching even a commented source listing for such references, and the assembly-language listing produced by MON or NRIBUG is not commented, and contains no symbolic labels or references of any kind. Finding such code references could become a nightmare. So how can we simplify the procedure?

This is where the hunt (h) command of NRIBUG proves to be very useful. For example, suppose you need to find all of the calls to CHRIN in the kernal jump table. You will first need to look up the exact byte sequence of this instruction. Figure 4-3 shows (an insert at the back of this book), the op-code for JSR as \$20. The address of the CHRIN entry is \$FFCF. Since addresses always appear low-byte first, the three bytes of this instruction will be 20 CF FF. Now, if the program occupies memory from \$8000 through \$9FFF, the resulting command to NRIBUG will be:

```
H 8000 9FFF 20 CF FF
```

NRIBUG or MON will respond with all addresses in the given memory range that begin this particular code sequence. Therefore, you can rapidly find them, both in memory and on your printed listing.

In the same way, you can locate a character string. It is usually not practical to search for a single byte, because there will be many matching bytes that will be found, even though they serve a different purpose. For example, if you Hunt for all JSR instructions (20), you will also be told of every ASCII space in character strings and every data byte that happens to be \$20. This does not help much.

Once you have located all references to a given location in memory, you can check each one to be sure that they are all valid, and that none are actually data that just happen to resemble code. Then you can make whatever adjustments you wish that are possible within the framework of the program.

Such modifications will generally be quite minor. Major modifications, such as rewriting a large block of code, are better performed on the source code which can then be reassembled. It is very difficult to perform such modifications without changing the length of the code, the locations of various labels within the program, or both.

DF00-DFFF	OPEN SLOT 2 (ALTERNATE DISK DRIVES)
DE00-DEFF	OPEN SLOT 1 (CP/M ENABLE)
DD00-DDFF	CIA 2 (SERIAL BUS, RS-232)
DC00-DCFF	CIA 1 (KEYBOARD)
D800-DBFF	COLOR RAM FOR VIDEO DISPLAY
D400-D7FF	SID (SOUND SYNTHESIZER)
D000-D3FF	VIC (VIDEO CONTROLLER)

Figure 4-2. I/O addresses in the C64.

Sector 7: Adding an Initialization Sequence

One thing that can be done with any program is to add an initialization sequence to the code. For example, you might wish to add some code to reset the printer, clear the screen, or perform some other preliminary task. Even though no place was made for such a routine in the original program, there are three possible ways to add it, depending on the structure of the program and its location in memory.

The first step, of course, is to design the needed code and list it on paper. Once we have this code, we can look for three possible places to put it: in front of the program, behind the program, or somewhere in the middle of the program. Putting code in the middle of a program may seem strange, but often an open buffer space is incorporated into a program. If the added code is executed before that buffer is ever used, and will not be executed again, then that buffer space can be used first for the code, and then as a buffer.

Of course, if the program loads at the high end of RAM, such as NRIBUG, then the obvious answer is to place the initialization routine in front of the existing code, and simply start with that code. By redefining the start of the program, we can place any code at all in front of the existing program, and simply re-save the modified program as a single unit.

But what if we have a program, such as FAS or PDSW, which will load at the very bottom of user RAM? Clearly we cannot add code to the start of such a program. What we can do, however, is place the extra code at the *end* of the program, and then put a JMP instruction to this code in place of the first instruction. The replaced instruction must, of

course, be duplicated at the end of the added code, and then followed by another JMP instruction back to the original code.

This last approach is quite common and works well, provided the added code will never be overwritten, or will never be needed after it has been overwritten by other data.

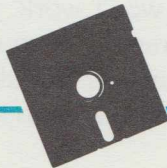
Sector 8: Modifying Compiled Object Code

If you have a program that turns out to be compiled from some high-level language, there is unfortunately very little that you can do to modify the code effectively. This is because the source code has essentially been turned into a long string of subroutine calls, with perhaps embedded data and/or character strings. If you have the source code available, but no compiler, you might be able to decipher which subroutine call or calls refer to what instruction, but even this is difficult.

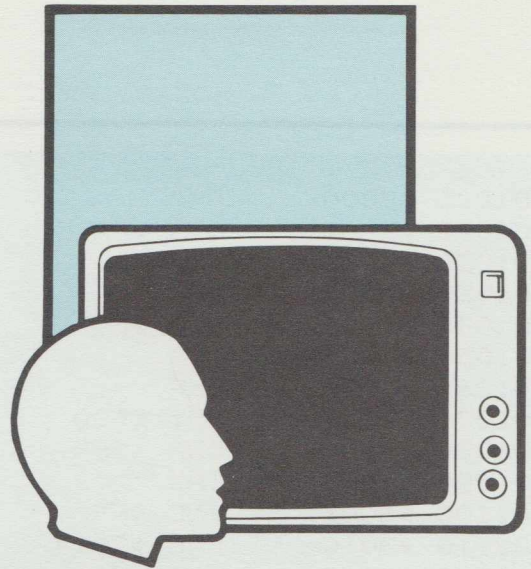
Without the source code, it is extremely difficult to figure out which subroutine does what, and what they all require as data. To see what we mean, try taking another look at the "HELLO, WORLD" program we compiled from BASIC in Module 6.

We strongly urge that you limit your modifications of such programs to the insertion of I/O initialization routines and similar functions, such as we have already discussed. Modifications to the program itself are best made to the source code, which can then be recompiled to form a new object program. The time and effort required to decipher and modify a compiled object program are excessive; it will cost you far less to purchase a compiler for that language and then buy, if possible, a copy of the source code. Even completely redesigning the program in source code will be much faster and easier than trying to make major modifications to the compiled code.

TRACK 5



SOLUTIONS AND PROBLEMS



Now that you have some idea of the kinds of changes that can be made to a program, and the different ways to find the places in the program where the changes are required, it is time to try your hand with an existing program and a practical modification. But before we start your new “homework” problem, let’s take a look at the problem from Module 7 and its solution.

Sector 1: The Problem from Module 7

In Module 7, we asked you to draw up a flowchart for the `DISPLAY-STRING` subroutine. Since we discussed this program in detail and also described the purpose of each segment of code, this should have been a fairly straightforward task.

Figure 5-1 shows the disassembly listing of this routine, as produced on the printer by `MON`. Following this listing and the discussion of the subroutine in Module 7, we developed the flowchart shown in Figure 5-2.

Note that the flowchart does not cover all of the details of program operation. What it does cover, as it should, is the logical flow of the program.

Sector 2: A Problem for Next Time

Earlier in this module, you worked with the program `PDSW`. Now complete the conversion from `IBM` to `Commodore`, if you have not already done so.

If you want an additional challenge while you await Module 9, try modifying the characters to reflect the graphics used by the `Commodore` computers. Keep in mind that each character on the screen is formed from an 8×8 dot matrix, with no blank dots between characters. This will require an adjustment to the 8×10 field presently used by `PDSW`. It will also change the number of character rows available on each page. You will need to determine how many character rows there may be, and how many characters can be permitted in each sideways row, to stay within memory limits.

One more possible addition to this program is to find a way to include the characters in reverse format, as they appear on the screen when you include cursor motion commands, color commands, etc., in a quoted string in a `BASIC` program. If you wish to try this in your program, there is a factor to watch out for: the screen display codes are

not the same as the ASCII character set. By using the screen display codes, you can specify reverse video merely by adding 128 (\$80) to the code. However, if you use the ASCII character set, this method won't work. Look up the available codes and characters in your User's Manual.

36

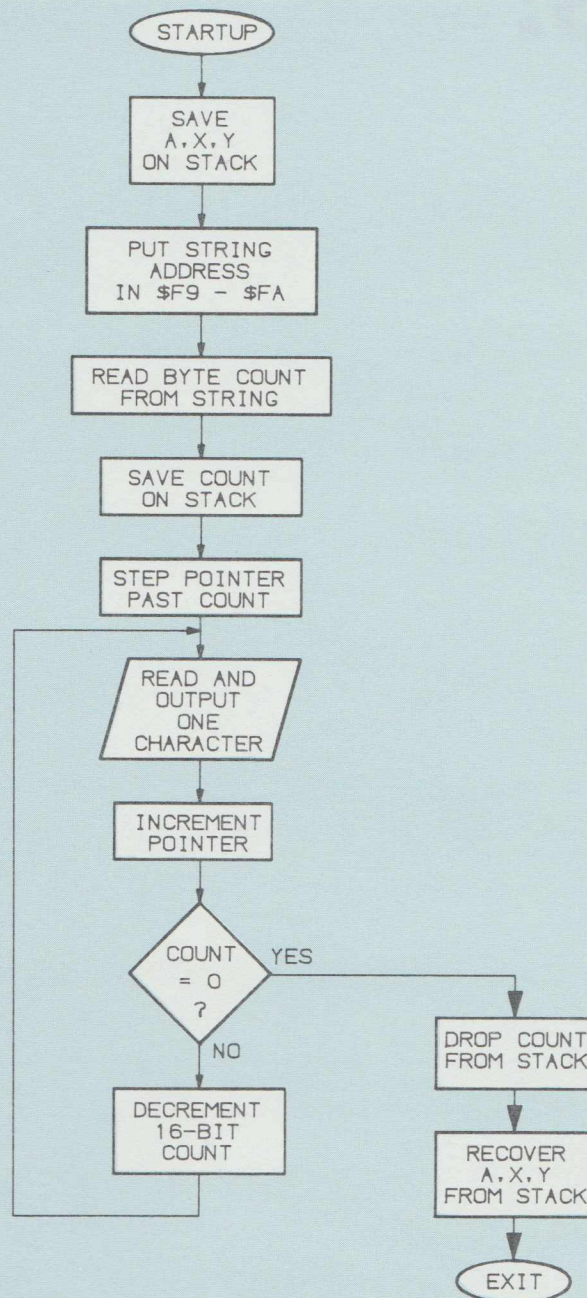
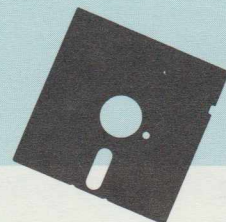


Figure 5-2. Flowchart of the DISPLAY-STRING subroutine.

NOTES:



APPENDIX A

PDSW SOURCE CODE

```
bell    equ    7
cr      equ    13
lf      equ    10
tab     equ    9
LINES   equ    48
eol     equ    '$'
esc     equ    27
;
;-----
;
LPRINT  MACRO  L1,L2,L3,L4,L5,L6      ;LPRINT A CHARACTER MACRO

IFNB    <L1>                          ;SET AL TO THE OPTIONAL PARAMETER
MOV     AL,L1
ENDIF
CALL    LPRINT1                      ;PRINT THE CHAR

IFNB    <L2>
LPRINT  L2,L3,L4,L5,L6
ENDIF

ENDM

;
;-----
;
FORM_FEED  macro
lprint  12
endm

;
;-----
;
SW_LF  macro
lprint  esc,'J',24,cr
endm

;
;-----
;
SW_SET_BIT  macro
lprint  esc,'K',cl,ch
endm

;
;-----
;
CSEG  SEGMENT byte public 'CODE'
ASSUME CS:CSEG,DS:CSEG,ES:CSEG
```



```

ORG      81H
fname    label    byte
ORG      100H
START    PROC     NEAR
        jmp      start1

        DB      CR,'Public Domain SideWays (pd)1985 by Donavon Kuhn ',cr,1f,1AH
;
handle    dw      0
char_flag db      0
end_of_file db     0
column    dw      0
row       db      0
;
start1:
        lea      dx,inst
        mov      ah,9
        int      21h                ;display banner

        xor      bh,bh
        mov      bl,ds:[80h]        ;get length of input filename
        lea      di,fname
        mov      byte ptr [di],0    ;make into an ASCIIZ string

;
        open     i,1,fname
        lea      di,fname-1        ;strip leading spaces
splp:
        inc      di
        cmp      byte ptr [di],' '
        jz       splp

        cmp      byte ptr [di],0    ;if no filename or '?',
        jz       usage              ;display usage message
        cmp      byte ptr [di],'?'
        jnz      no_usage

usage:
        lea      dx,usagestr
        jmp      error1

no_usage:
        mov      dx,di
        mov      ah,3dh
        mov      al,0
        int      21h                ;open the specified file
        mov      handle,ax
        jnc      no_open_err1
        jmp      error

```



```

no_open_err1:
    lea    dx,pausestr
    mov    ah,9
    int    21h                ;"press ESC to quit"

no_open_err:

    lea    di,line_ptr        ;fill text buffer with spaces
    mov    al,0
    cld
    mov    cx,LINES*1024+LINES*2
    rep    stosb

    mov    row,0              ;initialize pointers & counters
    lea    di,text_buffer
    mov    bx,0

lflp:
    inc    row
    cmp    row,LINES+1        ;reached max lines/page yet?
    jb     not_yet
    jmp    read_done

not_yet:
    mov    line_ptr[bx],di    ;pointer to the beginning of the line
    add    bx,2               ;don't wreck BX!
    mov    column,0

charlp:
    push    bx
    mov     bx,handle
    mov     ah,3fh             ;read char from file
    mov     cx,1
    lea     dx,inbuff
    int     21h
    pop     bx
    cmp     ax,1               ;was a character read?
    jz      read_ok

donejmp:
    mov     end_of_file,1      ;if not, file has ended
    jmp     read_done

read_ok:
    mov     al,byte ptr inbuff ;char in AL

    cmp     al,'Z'-64          ;if CTRL-Z,
    jz      donejmp           ;file is ended

    cmp     al,lf              ;if a new line,
    jz      lflp              ;go back to get next line

    cmp     al,cr              ;if CR, mark as null
    jnz     nocr
    mov     al,0
    jmp     place              ;place pointer in table

```



```

nocr:
    cmp    al,TAB        ;expand TAB characters
    jnz    notab         ;as spaces
    mov    al,' '

tablp:
    mov    [di],al
    inc    di
    inc    column
    call   colwidth
    test   column,7
    jnz    tablp
    jmp    charlp

notab:
    cmp    al,'L'-64      ;formfeed character?
    jz     read_done      ;if so, end of page

    cmp    al,32          ;ignore other control characters
    jb     spacectrl

place:
    mov    [di],al        ;put text character
    inc    di             ;in buffer
    inc    column
    call   colwidth       ;check for too wide
spacectrl:
    jmp    charlp

colwidth:
    cmp    column,1024
    jb     widthok
    lea    dx,toowide
    jmp    error1

widthok:
    ret

read_done:
    mov    ah,1           ;has a key been pressed?
    int    16h
    jz     read_done1

    mov    ah,0           ;if so, get it
    int    16h
    cmp    al,27          ;ESCape key?
    jne    read_done1

    jmp    all_done       ;if so, quit

```



```

read_done1:
    mov     cx,LINES*10           ;10 dot columns per character
    SW_SET_BIT                     ;printer to graphics mode

    mov     cx,LINES              ;line counter
    lea     si,line_ptr+(LINES*2)-2 ;pointer to this character line
    mov     char_flag,0

prlp:
    mov     al,' '                ;default is space
    mov     di,[si]
    cmp     di,0                  ;is pointer a null marker?
    jz      printit               ;if so, no more chars this line
    mov     al,[di]
    cmp     al,0                  ;is character a null?
    jnz     notzero               ;if so, rest of line is blank,
    mov     word ptr [si],0
    mov     al,' '                ;so output a space
    jmp     printit

notzero:
    inc     word ptr [si]         ;advance this pointer
    mov     char_flag,1          ;valid character flag

printit:
    sub     al,32                 ;skip ASCII control characters
    mov     ah,8                  ;index into bit table
    mul     ah
    lea     di,swbit              ;point to table
    add     di,ax                 ;point to correct entry

    push    cx
    mov     cx,8                  ;output 8 graphics columns

cxlp:
    mov     al,[di]               ;graphics print loop
    inc     di
    lprint
    loop    cxlp
    pop     cx

    lprint 0,0                    ;two nulls between rows

    sub     si,2
    loop    prlp

    SW_LF                         ;next printer line/char column

    cmp     char_flag,0           ;if all rows done,
    jz      next_page             ;advance to next page
    jmp     read_done

```


next_page:

FORM_FEED ;new printer sheet

cmp end_of_file,0 ;is file complete?
jnz all_done
jmp no_open_err ;if not, read 48 more lines

all_done:

mov ax,4c00h ;when done, exit
int 21h

;
;-----
;
LPRINT1 PROC NEAR ;LPRINT CHAR IN AL

push ax
push dx
XOR DX,DX
XOR AH,AH
int 17h
pop dx
pop ax
ret

LPRINT1 ENDP

;
;-----
;

inbuff db 0

inst DB 'Public Domain SideWays (pd)1985 by Donavon Kuhn '
db cr,lf,cr,lf,eol

toowide db 'Line width cannot exceed 1024 characters.',bell,cr,lf,eol

errmsg db 'File not Found or Open Error.',bell,cr,lf,eol

pausestr db 'Press ESC to Cancel',eol

error:

lea dx,errmsg

error1:

mov ah,9
int 21h
mov ax,4c01h ;error with errorlevel 1
int 21h

;
;

swbit: ;graphics data table

DB 0,0,0,0,0,0,0,0
DB 0,16,0,16,16,56,56,16
DB 0,0,0,0,0,36,36,36
DB 0,36,36,126,36,126,36,36
DB 0,24,124,2,60,64,62,24
DB 0,70,38,16,8,100,98,0
DB 0,118,136,136,86,48,72,48
DB 0,0,0,0,0,32,16,16
DB 0,16,32,64,64,64,32,16
DB 0,32,16,8,8,8,16,32

DB 0,0,68,56,254,56,68,0
 DB 0,0,16,16,124,16,16,0
 DB 32,16,16,0,0,0,0,0
 DB 0,0,0,0,126,0,0,0
 DB 0,16,16,0,0,0,0,0
 DB 0,64,32,16,8,4,2,0
 DB 0,60,98,82,74,70,66,60
 DB 0,124,16,16,16,80,48,16
 DB 0,126,66,48,12,2,66,60
 DB 0,60,66,2,28,2,66,60
 DB 0,28,8,254,72,40,24,8
 DB 0,60,66,2,2,124,64,126
 DB 0,60,66,66,124,64,32,28
 DB 0,16,16,16,8,4,66,126
 DB 0,60,66,66,60,66,66,60
 DB 0,56,4,2,62,66,66,60
 DB 0,16,16,0,0,16,16,0
 DB 32,16,16,0,0,16,16,0
 DB 0,8,16,32,64,32,16,8
 DB 0,0,126,0,0,126,0,0
 DB 0,16,8,4,2,4,8,16
 DB 0,8,0,8,4,2,66,60
 DB 0,60,64,94,82,94,66,60
 DB 0,66,66,126,66,66,36,24
 DB 0,124,34,34,60,34,34,124
 DB 0,28,34,64,64,64,34,28
 DB 0,120,36,34,34,34,36,120
 DB 0,126,34,40,56,40,34,126
 DB 0,112,32,40,56,40,34,126
 DB 0,30,34,78,64,64,34,28
 DB 0,66,66,66,126,66,66,66
 DB 0,56,16,16,16,16,16,56
 DB 0,56,68,68,4,4,4,14
 DB 0,99,36,40,48,40,36,98
 DB 0,126,34,32,32,32,32,112
 DB 0,65,65,65,65,73,85,99
 DB 0,66,66,66,70,74,82,98
 DB 0,24,36,66,66,66,36,24
 DB 0,112,32,32,60,34,34,124
 DB 0,3,60,74,66,66,66,60
 DB 0,114,36,40,60,34,34,124
 DB 0,60,66,2,60,64,66,60
 DB 0,28,8,8,8,8,73,127
 DB 0,60,66,66,66,66,66,66
 DB 0,8,20,34,65,65,65,65
 DB 0,54,73,73,73,65,65,65
 DB 0,65,34,20,8,20,34,65
 DB 0,28,8,8,8,20,34,65
 DB 0,127,33,16,8,4,66,127
 DB 0,120,64,64,64,64,64,120
 DB 0,2,4,8,16,32,64,128
 DB 0,120,8,8,8,8,8,120
 DB 0,0,0,0,130,68,40,16
 DB 255,0,0,0,0,0,0,0
 DB 0,0,0,0,0,8,16,16
 DB 0,63,66,62,2,60,0,0
 DB 0,46,49,49,46,32,32,96
 DB 0,60,66,64,66,60,0,0


```

DB      0,59,70,70,58,2,2,6
DB      0,60,64,126,66,60,0,0
DB      0,56,16,16,56,16,18,12
DB      124,2,62,66,66,61,0,0
DB      0,98,34,34,50,44,32,96
DB      0,56,16,16,16,48,0,16
DB      60,66,66,2,2,6,0,2
DB      0,38,40,48,40,36,32,96
DB      0,56,16,16,16,16,16,48
DB      0,73,73,73,73,118,0,0
DB      0,66,66,66,98,92,0,0
DB      0,60,66,66,66,60,0,0
DB      112,32,44,50,50,108,0,0
DB      14,4,52,76,76,54,0,0
DB      0,112,32,34,50,108,0,0
DB      0,124,2,60,64,62,0,0
DB      0,12,18,16,16,124,16,16
DB      0,58,70,66,66,66,0,0
DB      0,8,20,34,65,65,0,0
DB      0,54,73,73,73,65,0,0
DB      0,68,40,16,40,68,0,0
DB      124,2,62,66,66,66,0,0
DB      0,124,32,16,8,124,0,0
DB      0,12,16,16,96,16,16,12
DB      0,16,16,16,0,16,16,16
DB      0,48,8,8,6,8,8,48
DB      0,0,0,0,0,0,76,50
DB      0,127,65,65,34,20,8,0

```

```

;
;

```

```

line_ptr      label    word

```

```

usagestr:

```

```

db      cr,lf
db      'Usage:  DOSW filespec',cr,lf,cr,lf
db      'Where?  filespec is the name of a file to be printed',cr,lf
db      '          sideways to an Epson printer on LPT1:',cr,lf,eol

```

```

text_buffer    equ line_ptr+LINES*2

```

```

;
;

```

```

START      ENDP

```

```

CSEG      ENDS

```

```

END      START

```


6502 OP CODES

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
0x	BRK	ORA (zp,X)				ORA zp	ASL zp		PHP	ORA #	ASL A			ORA ad	ASL ad		0x
1x	BPL os	ORA (zp),Y				ORA zp,X	ASL zp,X		CLC	ORA ad,Y				ORA ad,X	ASL ad,X		1x
2x	JSR ad	AND (zp,X)			BIT zp	AND zp	ROL zp		PLP	AND #	ROL A		BIT ad	AND ad	ROL ad		2x
3x	BMI os	AND (zp),Y				AND zp,X	ROL zp,X		SEC	AND ad,Y				AND ad,X	ROL ad,X		3x
4x	RTI	EOR (zp,X)				EOR zp	LSR zp		PHA	EOR #	LSR A		JMP ad	EOR ad	LSR ad		4x
5x	BVC os	EOR (zp),Y				EOR zp,X	LSR zp,X		CLI	EOR ad,Y				EOR ad,X	LSR ad,X		5x
6x	RTS	ADC (zp,X)				ADC zp	ROR zp		PLA	ADC #	ROR A		JMP (ad)	ADC ad	ROR ad		6x
7x	BVS os	ADC (zp),Y				ADC zp,X	ROR zp,X		SEI	ADC ad,Y				ADC ad,X	ROR ad,X		7x
8x		STA (zp,X)			STY zp	STA zp	STX zp		DEY		TXA		STY ad	STA ad	STX ad		8x
9x	BCC os	STA (zp),Y			STY zp,X	STA zp,X	STX zp,Y		TYA	STA ad,Y	TXS			STA ad,X			9x
Ax	LDY #	LDA (zp,X)	LDX #		LDY zp	LDA zp	LDX zp		TAY	LDA #	TAX		LDY ad	LDA ad	LDX ad		Ax
Bx	BCS os	LDA (zp),Y			LDY zp,X	LDA zp,X	LDX zp,Y		CLV	LDA ad,Y	TSX		LDY ad,X	LDA ad,X	LDX ad,Y		Bx
Cx	CPY #	CMP (zp,X)			CPY zp	CMP zp	DEC zp		INY	CMP #	DEX		CPY ad	CMP ad	DEC ad		Cx
Dx	BNE os	CMP (zp),Y				CMP zp,X	DEC zp,X		CLD	CMP ad,Y				CMP ad,X	DEC ad,X		Dx
Ex	CPX #	SBC (zp,X)			CPX zp	SBC zp	INC zp		INX	SBC #	NOP		CPX ad	SBC ad	INC ad		Ex
Fx	BEQ os	SBC (zp),Y				SBC zp,X	INC zp,X		SED	SBC ad,Y				SBC ad,X	INC ad,X		Fx

= 1-byte immediate data
os = 1-byte offset from Program Counter
zp = 1-byte address in zero page
ad = 2-byte address in absolute memory

() = indirect addressing
,X = indexed by register X
,Y = indexed by register Y

Figure 4-3 Op codes and mnemonics for the 6502 microprocessor.